**605.202: Data Structures**

**James Rogers**

**Lab 3 Analysis**

**Due Date: April 23, 2019**

**Dated Turned In: April 27, 2019**

**Introduction:**

This project accepts a file containing a frequency table, a file containing input, a file to write the output to, and the type of method to use in processing the input. The project will output either an encoded or decoded version of the input, based off the requested method.

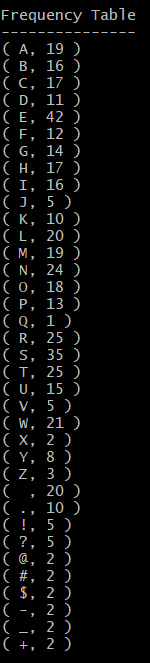
**Assumptions:**

I did an excessive amount of pre-processing on the provided FreqTable.txt. I didn’t want to alter any of the instructor-provided files, but the files contained several characters that would not render in my text editor (atom).

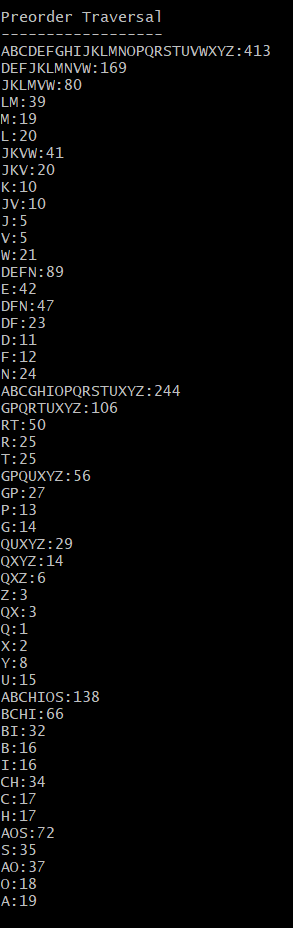
I also assumed that it was okay for me to cast input strings to uppercase due to the provided frequency table only having uppercase letters.

**Implementation:**

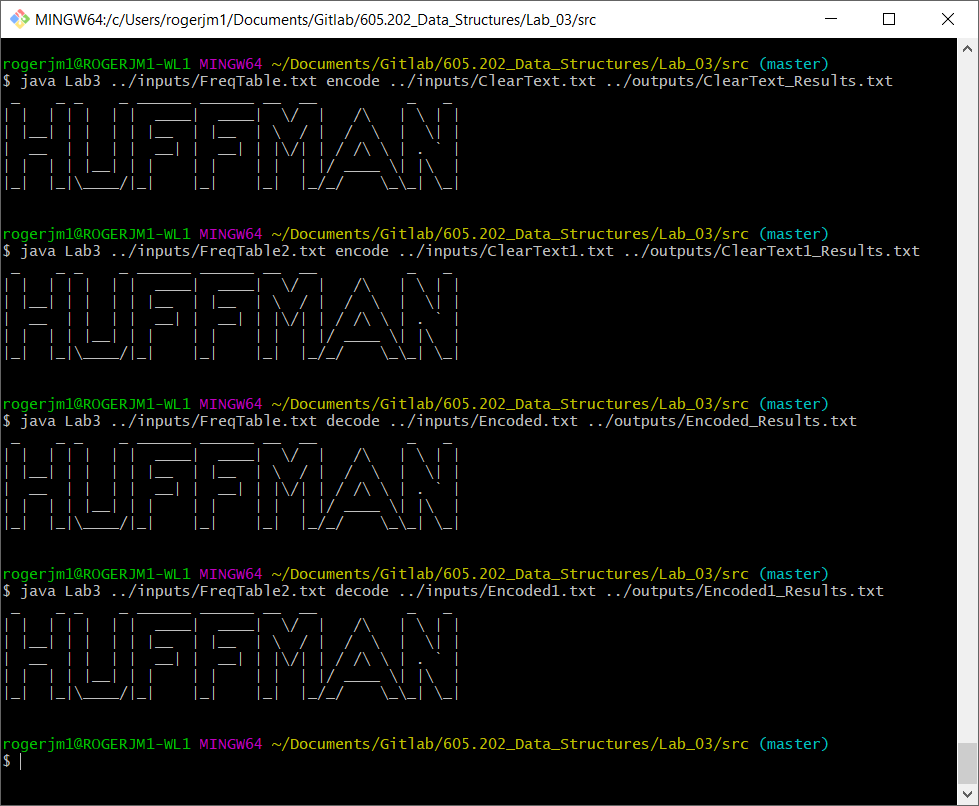
My implementation starts with handling the frequency table, for which I created a Frequency\_Table object. I didn’t want to risk the wrath of the grader by importing java.util.Dictionary, so I implemented the same functionality using two arrays stored as members of the class. This object is populated with the characters from the input table and their corresponding weights. A nice display of this is shown below:



Next, we have to construct the Huffman Encoding tree. I created two classes to handle this, a Tree class and a Leaf class. The Tree will read every element from the frequency table, create a baby leaf for each one, and then add them to the Plant Pot - a nice fertile place for the leaves to grow into a wonderful, large tree. The method for creating the tree is to locate the two smallest leaves in the plant pot, combine them into a new leaf that has the weight of both leaves combined, and place the newly created leaf back into the plant pot as the parent of the two smallest leaves – using the plantLeaf() method. Continue this process until only one Leaf object is remaining, and your tree is fully grown. The preorder traversal is shown below:



Lastly, we create a Coder object that we pass our fully grown Tree to. The Coder object contains methods to encode/decode strings based off the Tree provided in its constructor. Then, we continuously read a line from the provided input, process it, and write it out to the designated output file.

****

**Results:**

The provided strings:

Sally sells seashells by the seashore.�

Peter Piper picked a peck of pickled peppers a peck of pickled peppers Peter Piper picked.�

Houston, the Eagle has landed.�

Is that your final answer?

Encoded to:

11101111100010001101101111001000010001111011100101111111101101101000010001111011000101101100111011010111001011111111011011111101000010

10100010100101010001010011001101000101000101001100111010001000100110011111101000101101000100111100110110100110011101000100000101001100101000101010010100010100011101111110100010110100010011110011011010011001110100010000010100110010100010101001010001010001110101000101001010100010100110011010001010001010011001110100010001001100

11011111101011111101001111100111100111011010010111111010100010101101111111111000011111101110110001001100

1100111101001110111111110011011011111010111100001101110010111111110001111110111111000110101000

The provided strings:

01011001010110011111011011�

10110000101010011011101101100010110010101100010111000110111�

11111110001000111111101011111011001111111000100011111000001010000001110010111

Decoded to:

EIEIOH

ZEPHYRQZULU

ABRACADABRAMERLIN

Extended ASCII uses a full byte to encode a single character. To encode the string “Sally sells seashells by the seashore” would require 32 bytes or 256 bits – this is ignoring spaces and punctuation. The same process using our encoding scheme only required 134 bits, which is just over 50% of the required space of Extended ASCII. I don’t believe the tie-breaker scheme is very impactful on the overall number of bits required to encode a string, but it does make the tree noticably different. Two trees created with the same frequency table but with different tie-breaker criteria would encode and decode the same strings differently.

**Enhancements:**

I added the ability to process special characters if they are provided in the frequency table, including spaces and punctuation. The set of inputs demonstrating this are FreqTable2.txt, ClearText1.txt, and Encoded1.txt.

**Conclusion:**

Data compression, integrity, and storage is very important at this point in time. Law enforcement currently faces the issue of hard drives kept in evidence lockers for several years will have single bits flip and alter the hash of the entire drive, making it difficult to prove forensic integrity through lengthy appeals processes. Various companies that handle large amounts of network data can ingest gigabytes of pcap files in minutes.

This lab was very interesting in that it showed a non-traditional way of storing data can be significantly more efficient than the default. Additionally, I have a project that was looking at data compression for a unique system and we wanted to give different priorities to information that is seen frequently. I now know that this could be a frequency table with weights and I no longer have to vaguely describe what I’m thinking to people.